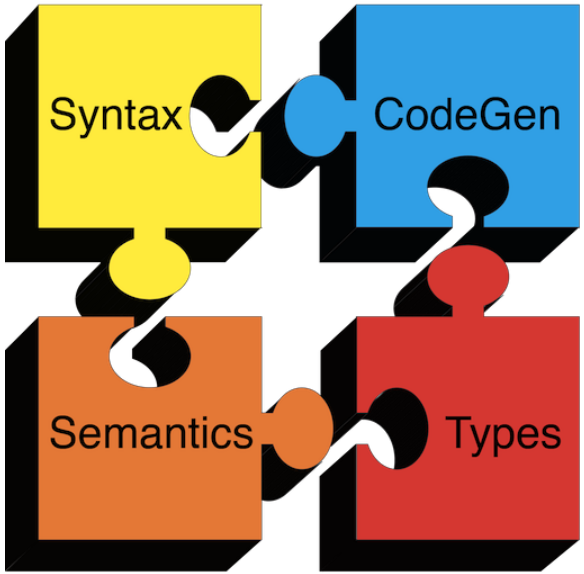

Compilers

Context-Free Grammars

CFGs

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

CFGs

- Programming languages have recursive structure

- An EXPR is
  if EXPR then EXPR else EXPR fi
  while EXPR loop EXPR pool
  …

- Context-free grammars are a natural notation for this recursive structure

CFGs

- A CFG consists of
  - A set of *terminals*
  - A set of *non-terminals*
  - A *start symbol*
  - A set of *productions*

CFGs

Productions can be read as rules.

CFGs

1. Begin with a string with only the start symbol S

2. Replace any non-terminal X in the string by the right-hand side of some production $X \rightarrow Y_1 \ldots Y_n$

3. Repeat (2) until there are no non-terminals

CFGs

CFGs

CFGs

Let *G* be a context-free grammar with start symbol S.
Then the language *L(G)* of *G* is:

CFGs

- Terminals are so-called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

A fragment of COOL

Some elements of the language:

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Simple arithmetic expressions

CFGs

Which of the strings are in the language of the given CFG?

- ☐ abcba
- ☐ acca
- ☐ aba
- ☐ abcbcba

$$
\begin{aligned}
S &\rightarrow aXa \\
X &\rightarrow \varepsilon \\
&\mid bY \\
Y &\rightarrow \varepsilon \\
&\mid cXc
\end{aligned}
$$

CFGs

The idea of a CFG is a big step. But:

- Membership in a language is “yes” or “no”; also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG’s (e.g., bison)

CFGs

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar